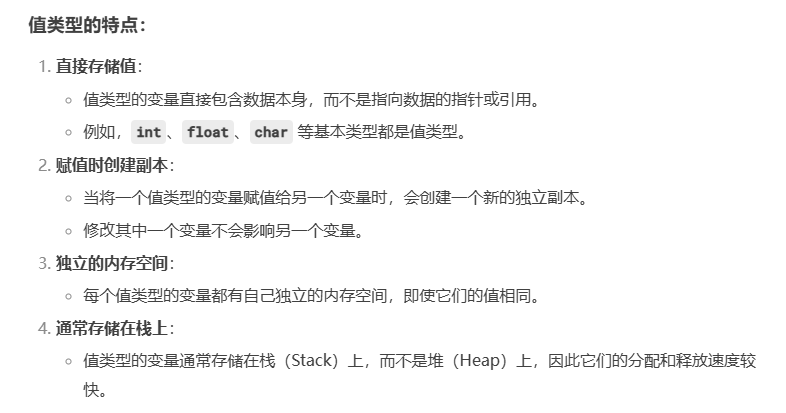
1. **缓冲区增长策略**：
   * 当字符串长度超过当前缓冲区大小时，std::string 会分配一个更大的缓冲区（通常是当前大小的两倍或下一个 2 的幂），并将原有内容复制到新缓冲区中。
   * 这种策略减少了频繁分配内存的开销，因为每次分配的空间比实际需要的更大，从而为未来的增长预留了空间。
2. **性能与空间权衡**：
   * **性能**：通过预分配更大的缓冲区，std::string 减少了频繁调用内存管理器的次数，使得追加操作的成本逐渐接近常数时间（O(1) 摊销复杂度）。
   * **空间**：这种策略的代价是可能会浪费一部分内存（例如，最多可能有一半的空间未被使用）。
3. **与 C 风格字符串的对比**：
   * C 风格字符串需要手动管理内存，容易出错（如忘记释放内存或缓冲区溢出）。
   * std::string 提供了更高的安全性和便利性，但可能会占用更多内存。
4. **符串的赋值与拷贝语义**：
   * 在 C++ 中，std::string 是值类型，赋值操作（如 s2 = s1）会创建一个新的独立副本，而不是共享内存。
   * 因此，修改一个字符串（如 s1[0] = 'n'）不会影响另一个字符串（s2 仍然是 "hot"）。
5. **字符串表达式的值语义**：
   * 字符串表达式（如 s2 + s3）会生成一个新的临时字符串对象，而不是直接修改现有字符串。
   * 例如，s1 = s2 + s3 + s4; 的执行过程如下：
     + 首先计算 s2 + s3，生成一个临时字符串。
     + 然后将 s4 连接到这个临时字符串，生成另一个临时字符串。
     + 最后将这个结果赋值给 s1，替换 s1 的原有值。
6. **内存管理的开销**：
   * 每次生成临时字符串时，都会动态分配内存。
   * 在赋值完成后，临时字符串和 s1 原有值的内存会被释放。
   * 这种频繁的内存分配和释放会导致大量调用内存管理器，可能影响性能。

字符串是值类型的。



写时复制

在 COW 字符串中，动态分配的存储可以在多个字符串之间共享。引用计数让每个字符串知道它是否在使用共享存储。当一个字符串赋值给另一个字符串时，只复制指针并增加引用计数。任何改变字符串值的操作都会首先检查是否只有一个指针指向该存储。如果有多个字符串指向该存储，任何可变操作（任何可能改变字符串内容的操作）都会分配新的存储并在修改之前复制字符串：

cpp

复制

COWstring s1, s2;

s1 = "hot"; // s1 是 "hot"

s2 = s1; // s2 是 "hot"（s1 和 s2 指向同一存储）

s1[0] = 'n'; // s1 在修改之前会创建其存储的新副本

1. **C++11 及更高版本的改进**：
   * C++11 引入了右值引用和移动语义，允许在某些情况下避免复制。
   * 例如，当传递一个临时字符串（右值）时，可以通过移动语义直接转移所有权，而不需要复制内容。

文本

描述已自动生成

S按值传递，导致每一次都会有临时对象创建，result和s的临时对象。如果缓冲区不够，还要进行复制。

图形用户界面, 文本, 应用程序

中度可信度描述已自动生成

修改了+=避免了临时对象，然后通过引用传入，更加避免了临时对象创建。但是通过引用传入并不一定会提示效率，每次调用都会解引用一个指针。

迭代器可以减少解引用带来提升，但并不一定是最优。

图形用户界面, 文本, 应用程序

描述已自动生成

容易被误用。

图形用户界面, 文本, 应用程序

描述已自动生成

极致性能用c去写，重写函数外加接口。

优化工作可能会达到这样一个点，即必须以简单性或安全性换取额外的性能增益。

第二次尝试优化字符串

使用更好的算法

图形用户界面, 文本, 应用程序

描述已自动生成

复制一个字符串子串，而不是单个字符

图形用户界面, 文本, 应用程序, 电子邮件

描述已自动生成

Append减少临时对象。

图形用户界面, 文本, 应用程序

描述已自动生成

原地操作进行移除。

文本

描述已自动生成

六种不修改标准库情况下优化字符串的方法

1，值传入值传出，可以优化为引用，但是可能会一些问题。

2，赋值运算符可以优化为字符串自带的函数，或者混合赋值，

3，对单个字符进行操作可以优化为子串操作。

4，c风格的语法去重写，可以避免函数调用。

5，更新算法，原地操作。

这段文本讨论了在C++编程中字符串转换的问题，特别是在不同类型的字符串之间进行转换时可能导致的性能损失。作者指出，不必要的字符串转换，特别是从C风格的字符串（以空字符终止的字符数组）到std::string的转换，可能会导致不必要的性能开销，因为这种转换涉及到字符的复制和内存的分配。

* 字符串的使用成本较高，因为它们是动态分配的，在表达式中表现为值，并且它们的实现需要大量的复制。
* 将字符串视为对象而不是值可以减少分配和复制的频率。
* 在字符串中预留空间可以减少分配的开销。
* 将字符串的常量引用传递给函数几乎与传递值相同，但可能更高效。
* 将结果字符串作为引用从函数中传出可以重用实际参数的存储空间，这比分配新的存储空间潜在地更高效。
* 一种仅有时减少分配开销的优化仍然是一种优化。
* 有时，一个不同的算法更容易优化或者本质上更高效。
* 标准库类实现是通用且简单的。它们不一定具有高性能，或者对任何特定用途来说都是最优的。